Security Lab – Buffer Overflow Attacks

Virtual Machine

This lab can be done with the **Ubuntu VM**. The remainder of this document assumes you are working with this VM.

It may be possible to do this lab on any x86 (64-bit version) system with installed *gcc* and *gdb*, but it has only been tested with the versions on the Ubuntu VM. It's therefore strongly recommended you do this lab using the Ubuntu VM.

1 Introduction

This lab consists of four main tasks. Task 1 is a step-by-step guideline to get familiar with analyzing and exploiting buffer overflow vulnerabilities and to learn about the GNU debugger (*gdb*), which is a very valuable tool in this context. The knowledge you acquire in task 1 will then be used in task 2, where you'll have to find and exploit a buffer overflow vulnerability in a program on your own. In addition, you'll analyze also the stack protection mechanisms of the GNU C compiler (*gcc*) in tasks 1 and 2. In task 3, you'll look at a different type of buffer overflow vulnerability that involves the heap (instead of the stack) at that allows an attacker to read sensitive information from memory. And finally, in task 4, you'll learn about format string overflows, which is yet another kind of buffer overflow vulnerability to get illegitimate read access to memory. This lab also serves to understand the possibilities and limitations of the buffer overflow countermeasures provided by the OS and *gcc* because they help to prevent exploitation of the vulnerabilities in tasks 1 and 2, but not in tasks 3 and 4.

The goal of this lab is that you get a profound understanding of the technical details of different types of buffer overflow attacks and that you can carry out some buffer overflow attacks on your own.

2 Basis for this Lab

- Download *bufferoverflow.zip* from Moodle.
- Move the file to an appropriate location (e.g., into a directory *securitylabs* in the home directory */home/user*).
- Unzip the file. This creates a directory *bufferoverflow* that contains the source code for both tasks.
- Turn off Address Space Layout Randomization (ASLR) during this lab so you don't have to deal with changing memory layouts during the different runs of the programs. To do this, enter the following in a terminal as *user* (followed by entering the password *user*):

```
$ sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

  To turn ASLR on again (don't do this for now), you can use the following command:

```
$ sudo bash -c 'echo 2 > /proc/sys/kernel/randomize_va_space'
```

  Note that turning off ASLR is not persistent, i.e., after rebooting Ubuntu, ASLR will be enabled again. To turn it off again, simply re-enter the first command.

3 Task 1 – Step-by-Step Guideline

In this task, you will exploit a buffer overflow vulnerability in a simple C program. Your main task is to understand what exactly happens during the following analysis. You find the source code (*secretfunction.c*) in folder *secretfunction* of the downloaded content. First, we discuss the relevant code sections.

The program consists of three functions: *main()*, *normal()* and *secret()*. *main()* is the entry point into the program and it first checks whether the ID of the current user is 0 (0 is the ID of user root). If that's

the case, i.e., if the program has been started by root, the function *secret()* is called. This function simply writes a string to the terminal and terminates the program.

```
void secret(void) {
  printf("Secret function was called!\n");
  exit(0);
}
```

If the program has been started by another user than root (meaning the ID is not 0), *secret()* is not called and *main()* prints the addresses of the two other functions to the terminal. These addresses could also be accessed by a debugger, but for simplicity they are printed directly to the terminal. Next, function *normal()* is called by *main()*. As a parameter (*args*), *normal()* receives a pointer to the string that was passed to the program (to *main()*) as a command line argument. *normal()* first creates a local buffer (*buff*) and fills this buffer with characters 'B' (using *memset()*). Then, it uses *strcpy()*, to copy the received string (*args*) into *buff*. Finally, the function prints the content, the address, and the size of the buffer to the terminal.

```
void normal(char *args) {
  char buff[12]; // allocate local buffer on stack

  memset(buff, 'B', sizeof (buff)); // fill buffer with B's
  strcpy(buff, args); // Copy received string to buffer
  printf("\nbuff: [%s] (%p)(%zu)\n\n", buff, buff, sizeof(buff));
}
```

As can clearly be seen, *normal()* does not check the length of the string *args* before it is copied into *buff*. If the string *args* is longer than what fits into *buff*, the remaining data is written beyond the end of the buffer. This means that *normal()* contains a buffer overflow vulnerability.

In the remainder of this task, we are going to exploit this buffer overflow vulnerability with the goal to call the function *secret()* without being root. To achieve this and to understand why this indeed is possible, the following step-by-step guideline is used.

Don’t be surprised if the memory addresses used on your system are different than the ones in this document. Depending on the used kernel, libc and compiler version, this may vary. You therefore have to use the addresses that correspond to the ones used on your system. Perform all steps in a terminal as *user*.

1. Delete any compiled components that may possibly be available on your system and compile the program, a corresponding *Makefile* is available:

   ```
   $ make clean
   $ make
   ```

2. Run the program and pass an arbitrary command-line argument with at most 12 characters:

   ```
   $ ./secretfunction ABCDEFG
   ```

   As you can see, the program works as expected. As the program has been started as a normal user and not as root, function *normal()* is called.

3. As a «proof-of-concept» of the vulnerability in *normal()*, run the program again and use a long command-line argument. This results in a segmentation fault, which happens if disallowed addresses are accessed during runtime and which clearly shows that a buffer overflow happened.

   To truly understand the buffer overflow vulnerability and to be able to exploit it to access the function *secret()* as a normal user, we have to analyze the structure and content of the stack. As

the vulnerability is located within *normal()*, we must analyze the stack briefly before the function is left. For this task, the GNU debugger (*gdb*) is well suited.

4. Start the debugger and pass the program as command-line argument:

```
$ gdb secretfunction
```

5. The command *list normal* display the source code in the area of the function *normal()*. *list normal, 27* displays the code of *normal()* until line 27.

```
(gdb) list normal, 27
20 void normal(char *args) {
21   char buff[12];
22
23   memset(buff, 'B', sizeof (buff));
24   strcpy(buff, args);
25   printf("\nbuff: [%s] (%p)(%zu)\n\n", buff, buff, sizeof(buff));
26 }
27
```

6. As we are interested in the stack right before leaving the function *normal()*, we must set a breakpoint at line 26, which causes the debugger to halt the program when it reaches that line.

```
(gdb) break 26
Breakpoint 1 at 0x81a: file secretfunction.c, line 26.
```

7. Now you can start the program. Use the string *AAA* as command line argument:

```
(gdb) run AAA
Starting program: home/user/securitylabs/bufferoverflow/
  secretfunction /secretfunction AAA
Address of secret(): (0x5555555547aa)
Address of normal(): (0x5555555547c4)

buff: [AAA] (0x7fffffffdc94)(12)

Breakpoint 1, normal (args=0x7fffffffe185 "AAA") at secretfunction.c:26
26    }
```

The program starts, and halts as expected at the breakpoint.

8. Now we can analyze the stack in detail:

```
(gdb) bt
#0  normal (args=0x7fffffffe185 "AAA") at secretfunction.c:26
#1  0x0000555555554888 in main (argc=2, argv=0x7fffffffddb8) at
      secretfunction.c:42
```

The debugger shows that there are two stack frames, one for each function that was called.

9. We are interested in the stack frame of the function *normal()*:

```
(gdb) info frame 0
Stack frame at 0x7fffffffdcb0:
 rip = 0x55555555481a in normal (secretfunction.c:26);
   saved rip = 0x555555554888
 called by frame at 0x7fffffffdce0
```

```
source language c.
Arglist at 0x7fffffffdca0, args: args=0x7fffffffe185 "AAA"
Locals at 0x7fffffffdca0, Previous frame's sp is 0x7fffffffdcb0
Saved registers:
 rbp at 0x7fffffffdca0, rip at 0x7fffffffdca8
```

Among other details, the output shows the value of the saved instruction pointer (`saved rip`[1]), which is `0x555555554888`. Note that the used Ubuntu VM is a 64-bit OS, which means that memory addresses have a length of 64 bits, but leading 0-bytes are in general not printed by *gdb*. This means the «true» 64-bit value of `rip` is `0x0000555555554888`. When leaving the function, this is the address where program execution will continue. In addition, in the last line of the output, you can see the addresses where the saved `rbp` (base pointer) and `rip` are stored in the stack frame.

10. We now analyze the content of the stack, starting from buffer `buff`. The address of `buff` was printed to the terminal (see step 7).

```
(gdb) x/8x 0x7fffffffdc94
0x7fffffffdc94:   0x00414141  0x42424242  0x42424242  0xffffdcd0
0x7fffffffdca4:   0x00007fff  0x55554888  0x00005555  0xffffddb8
```

This command shows eight (because of the '8' in *x/8x*) double words (a double word corresponds to 32 bits) starting from the specified address (instead of the address, one can also use a variable to display the double words starting from the address of the variable, so *x/8x buff* would have worked as well). We immediately see the following:

- The 1st double word contains the 3 A's (hexadecimal `0x41`, which is the ASCII code of A), which were passed to the program as an argument and which were copied into `buff`.
- The 6th and 7th double words (`0x55554888` and `0x00005555`) contain the saved `rip`, of which we know the value（`0x555555554888`）from the stack frame info above.
- The 4th and 5th double words (`0xffffdcd0` and `0x00007fff`) correspond to the saved `rbp`, of which we know the address (`0x7fffffffdca0`) from the stack frame info above.
- ***Important:*** The actual memory layout may look different on your system as it depends on the used version of *gcc*. It is therefore possible that there are additional double words between the end of `buff` and the saved `rbp`. If this is the case, consider this in the remainder of this task.

Note that in any double word displayed above (and below), the rightmost byte has the lowest and the leftmost byte the highest memory address. This is why the three A's in the first double word are shown as `0x00414141`: The first three bytes are A's and the 4th (the leftmost one and therefore the byte with highest address in this double word) is the null-byte (value `0x00`) that terminates a string in C and that was automatically appended to the entered A's.

Also, note how the 64-bit address of the saved `rbp` (`0x00007fffffffdcd0`) is spread «in opposite order» across the 4th and 5th double words: The reason is that we are using an x86 architecture (the currently dominating architecture in the «PC market») where the bytes of number types (such as addresses) are stored in little endian format. This means that the least significant byte (here `d0`) is stored at the lowest memory address (here `0x7fffffffdca0`) and the most significant byte (here `00`) at the highest memory address (here `0x7fffffffdca7`). The same holds for the saved `rip` in the 6th and 7th double words.

[1] With 16-bit OS, the instruction pointer was named ip, with 32-bit OS, it's eip, and with 64-bit OS, it's rip. The same holds for other registers, e.g., the base pointer: bp, ebp, rbp.

Based on the output of the command used above, the following illustration shows the relevant part of the stack even more detailed, including an indication of what is stored at the addresses (see column *Content*).

Address	Content	Bytes			
0x7fffffffdc94	buff	0x00	0x41	0x41	0x41
0x7fffffffdc98	buff	0x42	0x42	0x42	0x42
0x7fffffffdc9c	buff	0x42	0x42	0x42	0x42
0x7fffffffdca0	rbp	0xff	0xff	0xdc	0xd0
0x7fffffffdca4	rbp	0x00	0x00	0x7f	0xff
0x7fffffffdca8	rip	0x55	0x55	0x48	0x88
0x7fffffffdcac	rip	0x00	0x00	0x55	0x55
0x7fffffffddb0		0xff	0xff	0xdd	0xb8

In buffer `buff` we can see the null-terminated String *AAA*, which we have passed as an argument and which was copied into the buffer with *strcpy()*. One also sees that the buffer on the stack is filled from its starting address towards the higher addresses («downwards» in the illustration above), i.e., in the direction towards where `rbp` and `rip` are stored. The rest of the buffer contains B's (`0x42` hexadecimal, set by the function *memset()* as described above).

Next, there are 8 bytes for the saved base pointer (`rbp`) and 8 bytes for the saved return address (`rip`). Note again that in your case, there may be additional double words between the end of `buff` and `rbp`.

Now you can also see why the program is terminated with a segmentation fault if one uses a too long argument. The following illustration shows the stack when entering 12 A's (or more if there are additional double words between `buff` and `rbp`).

Address	Content	Bytes			
0x7fffffffdc94	buff	0x41	0x41	0x41	0x41
0x7fffffffdc98	buff	0x41	0x41	0x41	0x41
0x7fffffffdc9c	buff	0x41	0x41	0x41	0x41
0x7fffffffdca0	rbp	0xff	0xff	0xdc	0x00
0x7fffffffdca4	rbp	0x00	0x00	0x7f	0xff
0x7fffffffdca8	rip	0x55	0x55	0x48	0x88
0x7fffffffdcac	rip	0x00	0x00	0x55	0x55
0x7fffffffddb0		0xff	0xff	0xdd	0xb8

In this case, one of the bytes of the saved base pointer is overwritten with `0x00` (the null-byte that was automatically appended to the 12 A's entered by the user). As a result, the previously used stack frame cannot be regenerated in a correct way when the current method is left, which usually results in accessing disallowed addresses, which leads to the segmentation fault.

11. Now that we know in detail how the stack looks like, we can try to exploit the buffer overflow vulnerability in the function *normal()* to access *secret()* as a normal user. The idea is to abuse the vulnerability to overwrite the saved return address (`rip`) with the address of the function *secret()*. When returning from *normal()*, this results in executing *secret()* instead of returning to *main()*.

We already know the address of *secret()* from step 7: `0x5555555547aa`, which corresponds to the 64-bit value `0x00005555555547aa`. Leave the debugger using

```
(gdb) quit
```

and then start the program as follows (enter this in one line)

```
$ ./secretfunction AAAAAAAAAAAABBBBBBBB$'\xaa\x47\x55\x55\x55\x55
  \x00\x00'
```

The 12 A's fill the buffer, the 8 B's overwrite the base pointer (note that if there are additional double words between `buff` and `rbp` on your system, you have to insert four additional characters for each double word!) and the next 8 bytes (`\xaa\x47\x55\x55\x55\x55\x00\x00`) contain the address of the function *secret()*. Using `$'...'` instructs the (bash) shell to interpret the characters in hexadecimal format. Note that the bytes of the address of *secret()* have to be entered in reverse order (`\xaa\x47\x55\x55\x55\x55\x00\x00` for the address `0x00005555555547aa`) because little endian format is used (as explained above). The following illustration shows what happens on the stack:

Address	Content	Bytes			
0x7fffffffdc94	buff	0x41	0x41	0x41	0x41
0x7fffffffdc98	buff	0x41	0x41	0x41	0x41
0x7fffffffdc9c	buff	0x41	0x41	0x41	0x41
0x7fffffffdca0	rbp	0x42	0x42	0x42	0x42
0x7fffffffdca4	rbp	0x42	0x42	0x42	0x42
0x7fffffffdca8	rip	0x55	0x55	0x47	0xaa
0x7fffffffdcac	rip	0x00	0x00	0x55	0x55
0x7fffffffddb0		0xff	0xff	0xdd	0xb8

If you have done everything correctly, the function *secret()* will indeed be called and you should get an output as follows:

```
Address of secret(): (0x5555555547aa)
Address of normal(): (0x5555555547c4)

buff: [AAAAAAAAAAAABBBBBBBB...] (0x7fffffffdcf4)(12)

Secret function was called!
```

In this case, no segmentation fault happens although the base pointer was overwritten. The reason is that in *secret()*, the function *exit()* is called, which terminates the program. Therefore, the stack frame of the calling function (*main()*) is never regenerated during runtime.

12. In the lecture, you learned that compilers such as *gcc* can integrate protection mechanisms into the compiled code to detect some buffer overflow attacks. Here, we are using a current version of the *gcc* compiler, but the rather easy attack described above worked nevertheless. The reason is that the stack protection mechanisms of *gcc* were explicitly deactivated. You can see this by opening the *Makefile* and identifying the option *-fno-stack-protector* at the beginning of the file.

Remove the option (but keep option `-g`) and compile the program. Then call the program once with 12 and once with 13 A's. 12 A's should work but when using 13 A's, the program should terminate with the following message:

```
*** stack smashing detected ***: ./secretfunction terminated
```

It appears that *gcc* is indeed capable of detecting the buffer overflow attack. But how? You can understand this by again using *gdb* in the same way as above to analyze the stack. Enter 3 A's and look at the information about the stack frame of *normal()*:

```
(gdb) info frame 0
Stack frame at 0x7fffffffdcb0:
 rip = 0x555555554899 in normal (secretfunction.c:26);
   saved rip = 0x55555555491b
 called by frame at 0x7fffffffdce0
 source language c.
 Arglist at 0x7fffffffdca0, args: args=0x7fffffffe185 "AAA"
 Locals at 0x7fffffffdca0, Previous frame's sp is 0x7fffffffdcb0
 Saved registers:
  rbp at 0x7fffffffdca0, rip at 0x7fffffffdca8
```

This delivers us the value of the saved `rip`: `0x55555555491b`. Now look at the content of the stack starting at the address of `buff`:

```
(gdb) x/12x buff
0x7fffffffdc8c:   0x00414141  0x42424242  0x42424242  0xabf95900
0x7fffffffdc9c:   0x785ebfae  0xffffdcd0  0x00007fff  0x5555491b
0x7fffffffdcac:   0x00005555  0xffffddb8  0x00007fff  0x55554710
```

Here you can see 3 A's and the saved `rip`, but this time it is found at the 8th and 9th double words and not at the 6th and 7th as during the previous analysis. The 6th and 7th double words now corresponds to the saved `rbp` (the stack frame information above delivered us its address: `0x7fffffffdca0`). What's new are two additional double words starting at address `0x7fffffffdc98`. These are nothing else than a stack canary that was inserted by the compiler.

The resulting illustration of the stack including the stack canary looks as follows:

Address	Content	Bytes			
0x7fffffffdc8c	buff	0x00	0x41	0x41	0x41
0x7fffffffdc90	buff	0x42	0x42	0x42	0x42
0x7fffffffdc94	buff	0x42	0x42	0x42	0x42
0x7fffffffdc98	**stack canary**	**0xab**	**0xf9**	**0x59**	**0x00**
0x7fffffffdc9c	**stack canary**	**0x78**	**0x5e**	**0xbf**	**0cae**
0x7fffffffdca0	rbp	0xff	0xff	0xdc	0xd0
0x7fffffffdca4	rbp	0x00	0x00	0x7f	0xff
0x7fffffffdca8	rip	0x55	0x55	0x49	0x1b
0x7fffffffdcac	rip	0x00	0x00	0x55	0x55

With stack canaries, the compiler includes additional code. The code results in pushing a pseudo-random value (the canary) onto the stack when entering a function and in checking whether this

value is still the same right before leaving the function. This allows detecting buffer overflows that write beyond the «bottommost» variable on the stack, as this «destroys» the canary. Now it should be clear why the «attack» with 13 A's could be detected: The last A resulted in overwriting the first byte of the stack canary. You should also realize that it is no longer possible to use the attack described above, as it is not possible to overwrite the `rip` with the address of function *secret()* without destroying the canary.

Maybe you think that using 12 A's should be enough to overwrite the stack canary because of the null-byte that follows as the 13th character. But because *gcc* (at least in the used version) apparently sets the least significant byte of the canary to 0 (rightmost byte of the first canary double word in the representation above), the null-byte won't change the stack canary.

The value of the stack canary is newly created at every start of the program, so it's not possible for the attacker to predict its value. You can easily verify this by running the program several times and inspecting the value of the stack canary: it should have a different value each time.

4 Task 2 – Accessing a Secret File

This task serves to apply what you have learned above to another scenario. You find the source code and further files in folder *secretfile* of the downloaded content.

The scenario consists of a client and a server program. Compile the programs as follows and make sure to first delete any compiled components that may possibly be available on your system:

```
$ make clean
$ make
```

This results in two executables *client* and *server*. In addition, the files *public.txt* and *secret.txt* are copied to directory */tmp*. Start the server in a terminal (as *user*):

```
$ ./server
```

Open another terminal and start the client (also as *user*) by specifying the hostname where the server is running (*localhost*) and a (not too long...) message:

```
$ ./client localhost <message>
```

As a result of this, the client establishes a TCP connection to the server (using port 2222) and then sends the message to the server. The server receives the message and sends it back to the client together with the content of file */tmp/public.txt*. The client then prints the received data to the terminal.

4.1 Part 1: Finding and Exploiting the Vulnerability

Your task is to carry out a buffer overflow attack against the server to access the content of */tmp/secret.txt*. The idea is that by using the client, you send the server a specifically crafted message that causes the server to return the content of */tmp/secret.txt* to the client. Read the following hints before you start:

- First, study the source code *server.c* to understand how the server program works. Note: Constants such as *PORT*, *MSG_SIZE* and *BUF_SIZE* are defined in *common.h*.
- Try to find out where you could exploit a buffer overflow vulnerability to achieve the goal. Hint: The function *handleClientRequest()* looks interesting. The parameter *cfd* is a handle for the connection with the client. *recv()* serves to read data from the client and this data is copied into the buffer *message* in the first while loop. And *file* contains the value of *fpub*, which is a pointer to the string */tmp/public.txt*. Maybe this could be exploited somehow...
- To carry out the attack, you must understand how the involved variables are arranged on the stack, so you have to use the debugger as you have learned in task 1. In addition, the list of *gdb* com-

mands in the appendix may be helpful; it also includes some commands that were not discussed in task 1. In particular, *print* will likely be helpful to find out the addresses of *fpub* and *fsec*, which contain the paths of the files *public.txt* and *secret.txt* as strings.

Document your findings in the following box. In particular, describe the actual vulnerability and the steps you performed (and the intermediate results you got) to find out how it can be exploited. Also write down how the client program must be used for successful exploitation. Providing a good answer will get you the first lab point.

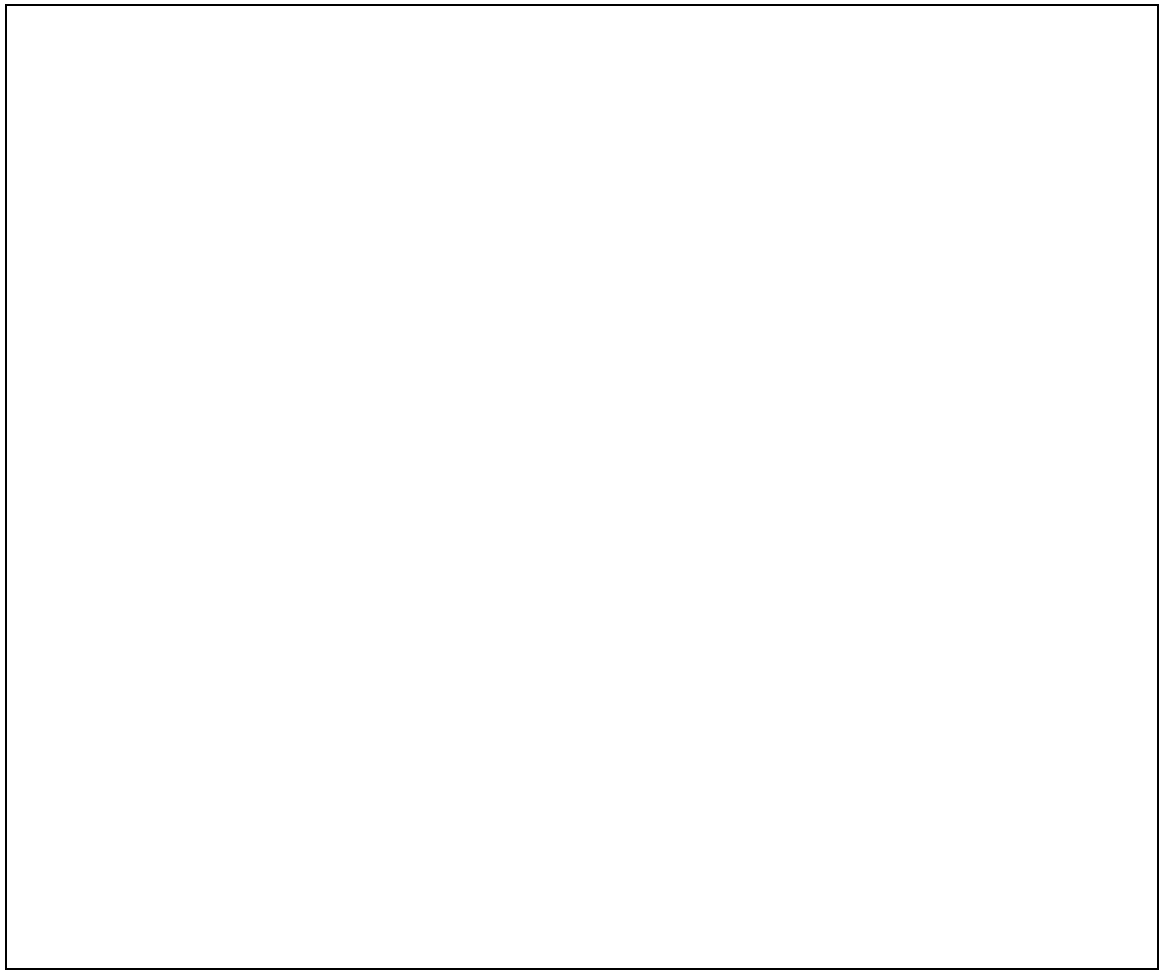
4.2 Part 2: Stack Protection

In this second part, we again look at the stack protection mechanisms offered by *gcc*. When inspecting the *Makefile* of this task, you can see the protection was again deactivated. Remove the option, compile the code and try again the same attack as before.

You'll see that the attack no longer works. You should now do the following:

- Analyze why the attack no longer works. Hint: Check whether the variables *file* and *message* are arranged differently on the stack than before and what influence this has on the attack.
- Is it possible to «make the attack working again» by slightly adapting the message that is sent to the server? If yes, do so. If no, explain why it can't be done, i.e., explain why the protection mechanisms effectively prevent this attack.

Document your findings in the following box (this is necessary to get the second lab point):



5 Task 3 – Heartbleed

In 2014, a serious vulnerability was detected in the OpenSSL implementation and got the name *Heartbleed bug* (CVE-2014-0160). OpenSSL is by far the most widely used TLS implementation and is employed by many popular software components such as the Apache web server. As a result of this widespread use, very many systems were affected. The implementation bug was introduced in the OpenSSL code in December 2011 and was released with OpenSSL 1.0.1 on 14th March 2012. OpenSSL 1.0.1g, released on 7th of April 2014 fixes the bug. This means that systems were exploitable during more than two years in the worst case. It is unclear whether and to what degree the vulnerability was exploited in the wild before it became publicly known.

In short, the vulnerability allows a remote attacker to read parts of the memory of TLS-protected server-side applications that are based on OpenSSL. It has been demonstrated that this allows accessing decrypted TLS data that was exchanged between clients and the server application (which may expose usernames, passwords, credit card numbers, sensitive e-mails, business-critical documents etc.) and, in some cases, even the private key of the server application (which is the holy grail for an attacker). This clearly underlines the criticality of the vulnerability. For more detailed information, refer to the links in the footnotes[2,3].

The vulnerability lies in the TLS heartbeat extension (RFC6520). The extension was introduced so a TLS endpoint can check if the peer is still alive. To do so, the endpoint sends a heartbeat request that

[2] General information about Heartbleed: *https://heartbleed.com*
[3] Technical information about the vulnerability: *https://stackabuse.com/heartbleed-bug-explained/*

includes some payload, and the other endpoint, assuming it is still alive, responds by sending a heartbeat response that includes the same payload.

The Heartbleed bug is a buffer overflow vulnerability but, as you will see, it's quite different compared to the vulnerabilities discussed in tasks 1 and 2. Therefore, it is well worth studying to improve your understanding of different types of buffer overflow problems. The goal of this task is that you understand the Heartbleed vulnerability and how to exploit it. To do this, you won't work with the real vulnerable OpenSSL code. Instead, you are using a program that implements a somewhat simplified version of the heartbeat protocol, but that contains more or less exactly the same vulnerability as the original OpenSSL code. The program consists of a client and a server component, where the client sends a heartbeat request to the server and the server responds with the corresponding heartbeat response. The program just uses plain TCP-based communication and no TLS, as the vulnerability lies in the incorrect handling of the data after it has been decrypted by the server. Therefore, the actual TLS protection measures have no influence on the vulnerability, so we can omit them for simplicity.

A heartbeat request packet consists of four fields. First, a packet header that contains the length of the remainder of the packet as an unsigned 4-byte int value (i.e., it contains the combined length of the payload length, payload and padding fields), then the length of the payload as an unsigned 2-byte int value, followed by the actual payload (which must be between 1 and 65'535 bytes), and optional padding data:

```
-------------------------------------------------------------
| packet header | payload length | payload         | padding    |
| (4 bytes)     | (2 bytes)      | (1-65535 bytes) | (optional) |
-------------------------------------------------------------
```

The payload and padding data together must not be longer than 65'535 bytes. As an example, if the 11-byte payload *testpayload* and the 11-byte padding *testpadding* are sent, the resulting request packet has a length of 28 bytes, the packet header contains the value 24, and the payload length contains the value 11.

The response uses basically the same format as the request, but the padding is not sent back:

```
-------------------------------------------------
| packet header | payload length | payload         |
| (4 bytes)     | (2 bytes)      | (1-65535 bytes) |
-------------------------------------------------
```

So, coming back to the example above, the resulting response has a length of 17 bytes, the packet header contains the value 13, and the payload length contains the value 11.

Maybe you are wondering why there are two length fields. The first length field (the packet header) basically corresponds to the TLS record header that is used by TLS to encapsulate the actual TLS content that is exchanged. The real TLS header contains more fields than just the length of the content (e.g., it contains also the type of the enclosed content), but in our case, just using the length is good enough. This length field is mainly used so the receiving endpoint knows when it has received the entire packet.

The second length field, the payload length field, is needed to clearly separate the payload from the padding in the request as only the payload is sent back in the response. But why is this padding field used anyway? This is included so that the heartbeat protocol can be used for additional purposes than just checking if the other endpoint is still alive. We are not going into further details[4] here as this is not relevant to understand the vulnerability. Of course, the response wouldn't need this payload length field as there's no padding, but the people defining the standard decided to include it nevertheless (its presence has no impact on the vulnerability).

[4] For the details, check out RFC 6520. The main reason why padding can be included is so that the heartbeat protocol can also be used for path MTU discovery.

With this information, you are now ready to use the simplified heartbeat program. You find the source code and further files in folder *heartbeat* of the downloaded content.

The two files *client.c* and *server.c* contain the code of the client and server programs. Common code used by both programs has been placed in *common.c*, and *common.h* contains some constants. Compile the programs as follows and make sure to first delete any compiled components that may possibly be available on your system:

```
$ make clean
$ make
```

This results in two executables *client* and *server*. Start the server in a terminal (as *user*):

```
$ ./server
```

Open another terminal to run the client (also as *user*). This is done by specifying the hostname where the server is running (*localhost*), the length of the payload, the actual payload, and optionally the padding:

```
$ ./client localhost <payload length> <payload> {<padding>}
```

As a test, use the client as follows:

```
$ ./client localhost 11 testpayload testpadding
```

This establishes a TCP connection to the server (using again port 2222 just like in task 2), sends the heartbeat request and gets back the heartbeat response. Looking at the outputs in the terminals, you can see that the client sends a request of length 28, receives a response of length 17, and that it receives the payload the payload *testpayload* that was included in the request. Omitting *testpadding* when using the client sends and receives 17 bytes, as no padding is included in the request.

Before analyzing the program and understanding the vulnerability, let's briefly talk about the buffer overflow countermeasures provided by the OS and *gcc*: They are of no help against the Heartbleed vulnerability. Correspondingly, the protection measures of *gcc* are not deactivated in this task (so the option *-fno-stack-protector* is not used in the *Makefile*). Also, you could turn on ASLR again as it won't help to protect from the attack. However, during program analysis, it's best to leave ASLR turned off as this means the memory addresses will be the same during each run, which makes analysis easier. But at the end of this task, once you understand the vulnerability and have managed to exploit it, feel free to turn ASLR on again using the command described at the beginning of this document and verify that the attack indeed still works.

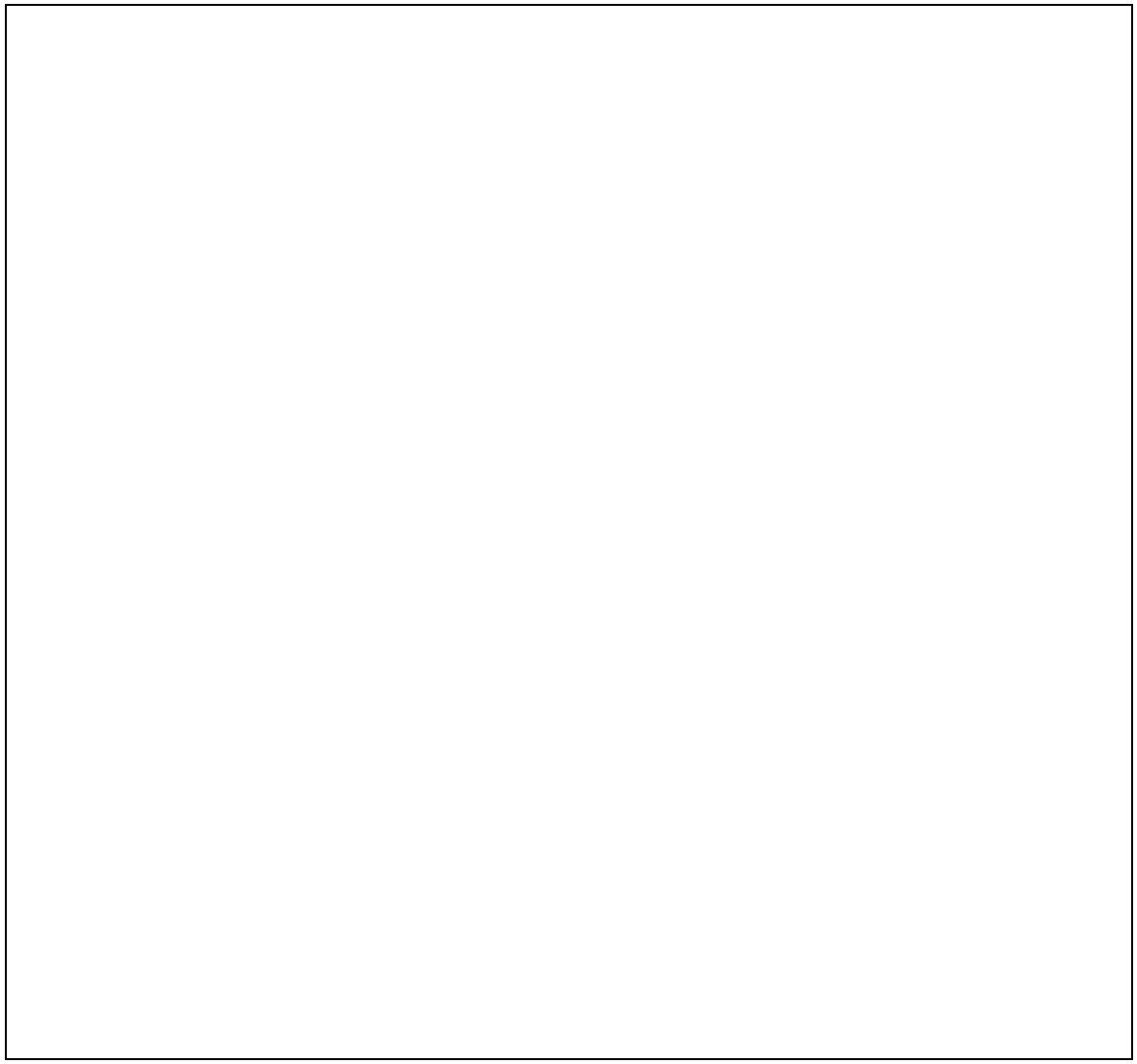
Next, study the programs. You don't have to study everything as a lot of the code is needed to set up the sockets and to implement sending and receiving requests and responses, which is all not directly related to the vulnerability. Therefore, focus your analysis on the following code sections in *server.c*:

- At the beginning of *main*, a function *simulateSensitiveData* is called. Looking at this function, you can see that this allocates memory on the heap (using *malloc*) and copies about 600 bytes of data into the allocated memory. This simulates a few previous HTTP requests that were processed by the server, i.e., assume they were received from the browser of another user (TLS-protected), then they were decrypted and stored on the heap, and then they were passed to the application layer for processing. In the end, by exploiting the Heartbleed vulnerability, you should get access to this information (which includes interesting details such as session IDs, a username / password pair, and information about a credit card) by using the client program. When returning from the method, the allocated memory is freed (line *free(simData)* in *main*) because we assume the requests have been completely processed – so freeing this memory would certainly be done by a real TLS server to prevent memory leaks.

- Requests are handled in a while loop in the *main* function. When a client connects, the request is handled in the main process or in a separate process, depending on the setting of *FORK* in *common.h*. Per default, *FORK* is set to 1 (true) and in this case, the *fork* function is used to handle the request in a separate process (which is an exact copy of the main process), while the main (parent) process waits for the next connection. This forking of the process is not related to the vulnerability. The main reason why we are handling a request in a separate process is to make sure that whenever a client connects and its request is processed, the initial memory situation on the heap is always exactly the same, i.e., it always corresponds to the situation right after the sensitive data has been stored on the heap and the memory has been freed again. This guarantees reproducibility whenever a request is handled without having to restart the server. The problem with forking is that it makes debugging with *gdb* cumbersome. Therefore, when debugging the server with *gdb*, you should first set *FORK* to 0 (false) and rebuild the program with *make*. The disadvantage of this is that unambiguous reproducibility when using multiple requests is no longer guaranteed, so you may have to restart the server between analyses of individual requests.
- To handle a request (either in the main or the child process), the *readRequest* function is used first to read the heartbeat request from the client. If this is successful, the function returns a pointer to the memory area where the request has been stored and passes this to the *sendResponse* function, which sends the heartbeat response to the client. In the remainder, focus on these two functions.

Your task is now to find and understand the vulnerability and, by using the client program, to exploit the vulnerability so you can get access to the sensitive data stored on the heap. To give you some hints: Carefully check where the memory to store the request in *readRequest* is allocated, at what address this memory is allocated, and how this relates to the address of the memory that was allocated in *simulateSensitiveData* (these addresses can be analyzed with *gdb* (after setting *FORK* to 0), or by using *printf("%p\n", ptr)* to print the address of a pointer *ptr* to the terminal). Also, check how much data from the stored request is actually sent back to the client in the response.

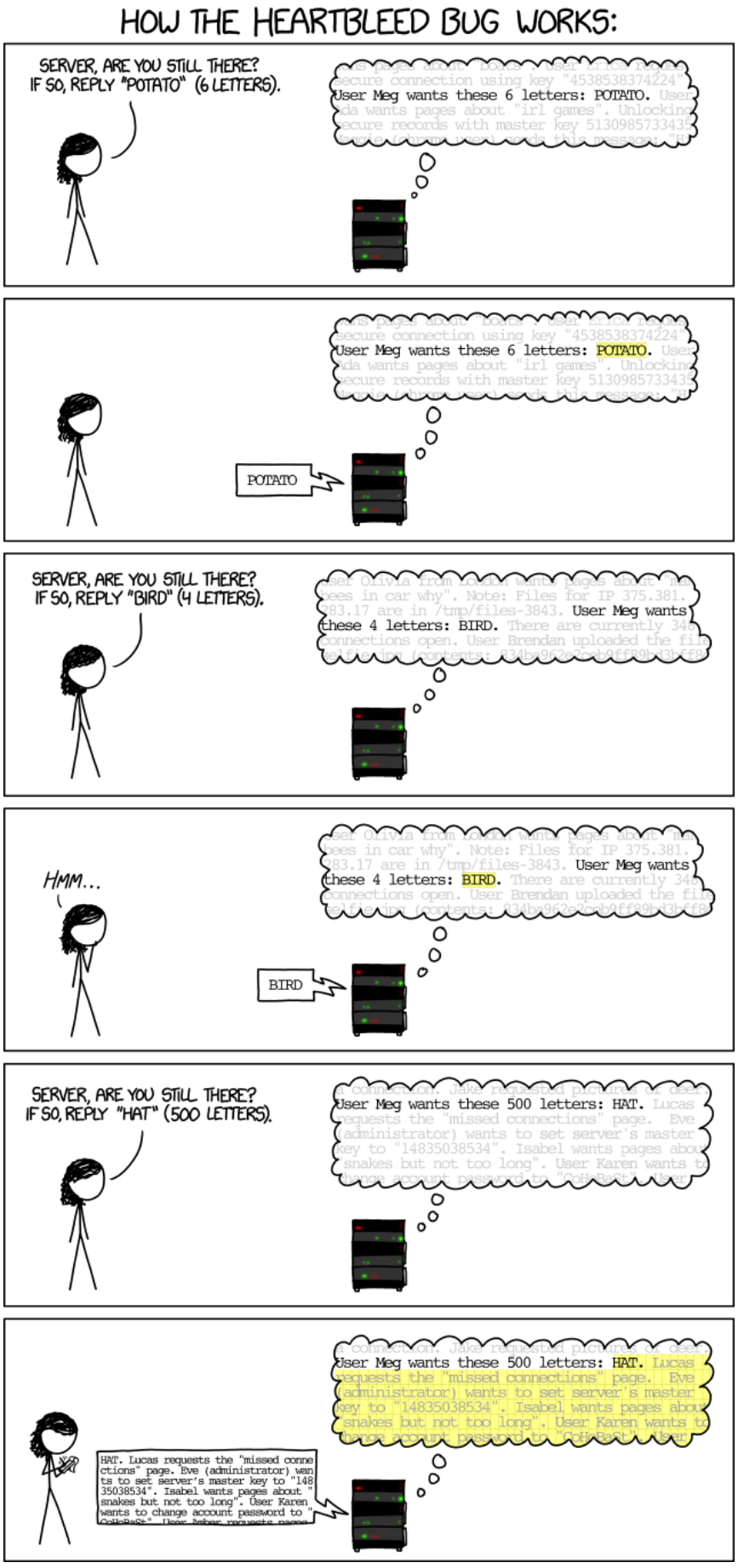
Document your findings in the following box. In particular, describe the vulnerability with all relevant details so it can clearly be understood, and how the client can be used (by giving a specific example) to exploit the vulnerability. Providing a good answer will get you the third lab point.



Let's summarize this task: Here, you have encountered quite a different type of buffer overflow vulnerability than what you have seen in the first two tasks, as the vulnerability is not used to *write* memory locations beyond the end of a buffer, but to *read* them. This is also sometimes identified as a read buffer overflow. They are less well known than overflows related to writing, but can have a major security impact as can easily be seen here. Another lesson from this task is that buffer overflows not only affect the stack, but also the heap. In the end, memory is memory and if the code allows to read or write beyond the end of allocated memory, data that should not be accessible may be read or written by an attacker, no matter whether this happens on the stack or the heap. Fixing the vulnerability – as is often the case with buffer overflow vulnerabilities – is not difficult and merely requires checking that the payload length field in the request is nor greater than the combined length of the payload and the padding. If that is done correctly, no overflow will be possible.

This vulnerability also shows you that against some buffer overflow issues, the protection measures by the OS and compilers are of no help. The attack always works, no matter what the exact addresses of the memory space allocated on the heap are. Also, nothing is overwritten on the stack, so stack canaries cannot help, and as the attack happens on the heap, reordering local variables on the stack obviously won't have any benefit either.

A discussion of the Heartbleed bug would not be complete without having a look at the famous xkcd comic (*https://xkcd.com/1354/*), which illustrates Heartbleed with great clarity and simplicity.


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
these 4 letters: BIRD.
HMM...
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to

6 Task 4 – Format String Overflow

In the final task, another type of buffer overflow is discussed, so-called format string overflows. Corresponding vulnerabilities can happen if *printf* (and similar functions such as *fprintf*, *sprintf*, etc.) is used incorrectly.

printf is used to write data to *stdout* (often the terminal). *printf* is a function that supports a variable number of parameters and that requires at least one parameter. The following shows a typical usage of *printf*:

```
char name[] = "John";
int age = 25;
printf("Hello %s, you are %d years old.\n", name, age);
```

The first parameter of *printf* is the format string. It uses placeholders (format tags) that start with *%*. *%s* means that the placeholder should be replaced with a string (in C a null-terminated char array) and *%d* means the same for an int. There are many other format tags supported to handle various data types. The actual values to be used for the placeholders follow in the additional parameters, in the right order. So, in this case, *%s* will be replaced with the value of *name*, which is the string *John*, and *%d* will be replaced with the value of *age*, which is 25. Therefore, the following will be written to *stdout*:

```
Hello John, you are 25 years old.
```

Now what happens behind the scenes. From the lecture, you know that when calling a function, the function parameters are passed via registers (specifically, via six registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`). The called function stores the content of these registers on the stack, right on top of the local variables (assuming the stack is illustrated «upside-down» with higher addresses at the bottom). The order in which the function parameters are stored on the stack is the same as in the parameter list of the function. So, when calling *printf* as in the example above, a pointer to the char array that contains the format string is first stored on the stack (as you certainly know, C always passes the pointer (the starting address) to the array whenever an array is used as a function parameter). Next, a pointer to the char array *name* (that contains the string *John*) is stored on the stack. Finally, the value of *age* (25) is stored on the stack. Therefore, the stack of function *printf* may look as illustrated below after the function parameters have been stored (we assume a 64-bit system, so every row contains 8 bytes; also note that the addresses are fictional and only for illustrative purposes):

Address	Content
`0x7fffffffd020`	`parameter 3: value of age (25)`
`0x7fffffffd028`	`parameter 2: address of char array name`
`0x7fffffffd030`	`parameter 1: address of format string`
`0x7fffffffd038`	`local variable 2 of printf`
`0x7fffffffd040`	`local variable 1 of printf`
`0x7fffffffd048`	`stack canary`
`0x7fffffffd050`	`stored rbp of calling function`
`0x7fffffffd058`	`return address (stored rip)`

Note that we assume that *printf* also uses two local variables (we don't care about them, they are just included to show that the function parameters are stored on top of the local variables). Below the local variables, there are – as usual – the stack canary, the stored `rbp`, and the return address.

To create the output string, *printf* now does the following: It accesses the first function parameter (the address of the format string) and evaluates the format string at this address. In this format string, it sees the first format tag *%s*. As a result of this, it accesses the memory location of the second function parameter (the address of the char array *name*, which is located 8 bytes before («above») the first parameter) to get the corresponding string. And then, it evaluates the format string further and sees the second format tag *%d*. As a result of this, it accesses the memory location of the third function parameter, which must be located 8 bytes before the second parameter, and interprets its content (the lower 4 bytes) as an int value (25). Now, the values of all placeholders have been determined and the complete string is written to *stdout*.

Now let's assume *printf* is used as follows (*gcc* will create a warning, but the code compiles successfully):

```
printf("Hello %s, you are %d years old.\n");
```

In this case, *printf* contains only one parameter and correspondingly, only one parameter – the address of the format string) will be placed on the stack and the resulting the stack looks as follows:

Address	Content
0x7fffffffd020	undefined (interpreted as parameter 3)
0x7fffffffd028	undefined (interpreted as parameter 2)
0x7fffffffd030	parameter 1: address of format string
0x7fffffffd038	local variable 2 of printf
0x7fffffffd040	local variable 1 of printf
0x7fffffffd048	stack canary
0x7fffffffd050	stored rbp of calling function
0x7fffffffd058	return address (stored rip)

But during runtime, *printf* does not know that the additional parameters are missing, and it does the same as in the first case above: It evaluates the format string and accesses the memory locations where the second and third function parameters would be stored to get the string and the int value. Depending on the actual data stored at these locations (this cannot be predicted, so *undefined* is used in the illustration above), a garbled string may be printed, or a segmentation fault may occur.

What was demonstrated here is a *format string overflow*, as the format string was abused to access memory locations (here the locations where the second and third parameters would be stored) that should not be accessed legitimately. It is debatable whether such an overflow should be identified as a buffer overflow as no actual reading beyond the end of a buffer (in the sense of an array) is done, but most articles and textbooks count this as a buffer overflow as the function parameters on the stack basically correspond to a «buffer» that is read beyond the intended memory locations.

Above, it was mentioned that six registers are used to pass parameters to a function. But what if more than six parameters are used? In this case, the additional parameters are directly passed on the stack. To illustrate this, consider the following example:

```
char text[] = "square numbers";
int sq1 = 1, sq2 = 4, sq3 = 9, sq4 = 16, sq5 = 25, sq6 = 36, sq7 = 49;
printf("The %s are: %d, %d, %d, %d, %d, %d, %d\n", text, sq1, sq2,
        sq3, sq4, sq5, sq6, sq7);
```

This will write the following to *stdout*:

```
The square numbers are: 1, 4, 9, 16, 25, 36, 49
```

In this case, as there are nine parameters to *printf*, only the first six will be passed using registers and the additional three are passed directly on the stack. Once the function is called and the parameters have been stored, the stack looks as follows:

Address	Content
0x7fffffffd008	parameter 6: value of sq4 (16)
0x7fffffffd010	parameter 5: value of sq3 (9)
0x7fffffffd018	parameter 4: value of sq2 (4)
0x7fffffffd020	parameter 3: value of sq1 (1)
0x7fffffffd028	parameter 2: address of char array text
0x7fffffffd030	parameter 1: address of format string
0x7fffffffd038	local variable 2 of printf
0x7fffffffd040	local variable 1 of printf
0x7fffffffd048	stack canary
0x7fffffffd050	stored rbp of calling function
0x7fffffffd058	return address (stored rip)
0x7fffffffd060	parameter 7: value of sq5 (25)
0x7fffffffd068	parameter 8: value of sq6 (36)
0x7fffffffd070	parameter 9: value of sq7 (49)

Behind the scenes, this works as follows: If a function calls *printf* and wants to pass more than six parameters, it places the first six parameters in the registers used for function parameters, in the usual way. In addition, the remaining parameters are placed on the stack, in opposite order. Therefore, in the current case, the calling function first places parameter 9 on the stack, then parameter 8, which is followed by parameter 7. Note that this happens in the stack frame of the calling function, on top of the local variables of the calling function. Then, *printf* is called. *printf* creates its own stack frame by first storing the return address, the `rbp` of the calling function, and the stack canary on the stack. Then, space for the local variables of *printf* is created and next, the first six parameters from the registers are copied into the stack frame of *printf*. With the additionally parameters, no copying or storing is needed as they are already on the stack. This means that *printf* will directly access parameters 7, 8 and 9 in the stack frame of the calling function if it wants to use them.

Once more, assume *printf* is used by only specifying the format string, but no additional parameters:

```
printf("The %s are: %d, %d, %d, %d, %d, %d, %d\n");
```

Again, there's only one parameter will be placed on the stack, which looks as follows:

Address	Content
0x7fffffffd008	undefined (interpreted as parameter 6)
0x7fffffffd010	undefined (interpreted as parameter 5)
0x7fffffffd018	undefined (interpreted as parameter 4)
0x7fffffffd020	undefined (interpreted as parameter 3)

0x7fffffffd028	undefined (interpreted as parameter 2)
0x7fffffffd030	parameter 1: address of format string
0x7fffffffd038	local variable 2 of printf
0x7fffffffd040	local variable 1 of printf
0x7fffffffd048	stack canary
0x7fffffffd050	stored rbp of calling function
0x7fffffffd058	return address (stored rip)
0x7fffffffd060	parameter or local variable in stack frame of calling function (interpreted as parameter 7)
0x7fffffffd068	parameter or local variable in stack frame of calling function (interpreted as parameter 8)
0x7fffffffd070	parameter or local variable in stack frame of calling function (interpreted as parameter 9)

Parameters 2-6 will behave similar as in the example above with three parameters. However, when *printf* accesses the locations where parameters 7-9 should be, it actually accesses the stack frame of the calling function, and probably accesses function parameters or local variables that are stored there. To summarize: Whenever *printf* is called with a format string that contains more than five format tags, it not only accesses memory locations in the own stack frame (for the first six parameters), but also memory locations in the stack frame of the calling function and below (for further parameters). And it doesn't matter whether the parameters are actually passed to *printf* or not, the corresponding memory locations will always be accessed.

These two format string overflow examples discussed so far are likely «only bugs» (unless the memory locations that are accessed happen to contain sensitive data). However, if the attacker can choose the format string to be used, very interesting attacks are possible.

With this knowledge, you should be ready to find and exploit a format string overflow vulnerability in a program. The corresponding program *login.c* and its *Makefile* are located in folder *login* of the downloaded content.

Just like in task 3, the buffer overflow countermeasures provided by the OS and *gcc* won't be of any help against this vulnerability. Therefore, stack protection is not deactivated in the *Makefile*. However, it's again recommended to have ASLR turned off during analysis so the addresses will be the same during each program invocation.

Build and run the program in a terminal (as *user*):

```
$ make clean
$ make
$ ./login
```

The program simulates a system login with username and password. Try it out by entering valid credentials (e.g., username *root* and password *master*), which gives you terminal access (this is a «dummy» terminal and the only command it understands is *exit*). Entering *exit* brings you back to the login. If you enter invalid credentials (use anything you like), login won't be accepted.

Next, study the program. You shouldn't have problems to understand the code. The outer *while* loop of the *main* function calls *doLogin* to perform the login process. *main* also stores the valid username / password combinations in the arrays *usernames* and *passwords*. Hardcoding the credentials is of course not a smart idea and as a result of this, the scenario may not be very realistic, but it serves well to demonstrate that a format string vulnerability can be exploited to basically read any content stored

on the stack. The two arrays *usernames* and *passwords* are passed to the *doLogin* function and *doLogin* only returns 1 (true) if the entered username / password combination is valid. As long as *doLogin* returns 0 (i.e., login failed), the outer *while* loop in main will simply repeatedly call *doLogin* so the user can try again. If login is successful, the inner *while* loop in main is entered, which simulates terminal access after a successful login. Entering *exit* leaves the inner *while* loop and *doLogin* is called again.

Once more, your task is to find and understand the vulnerability – in this case a format string vulnerability – and to exploit it. The goal of the exploitation is to get the following information by using the login functionality of the program:

- **All usernames and passwords**, i.e., the complete content of arrays *usernames* and *passwords* that are stored in function *main*.
- **The value of the base pointer (`rbp`) of the stack frame of the main function**. This value is stored at the bottom of the stack frame of *doLogin* when the program is running within *doLogin*.
- **The used stack canary value**. Note that no canary is used in the stack frame of function *doLogin* (as *doLogin* does not have any local array variables, so there's nothing to overwrite), but the canary is used in the stack frame of function *main*.

Take the following hints into account:

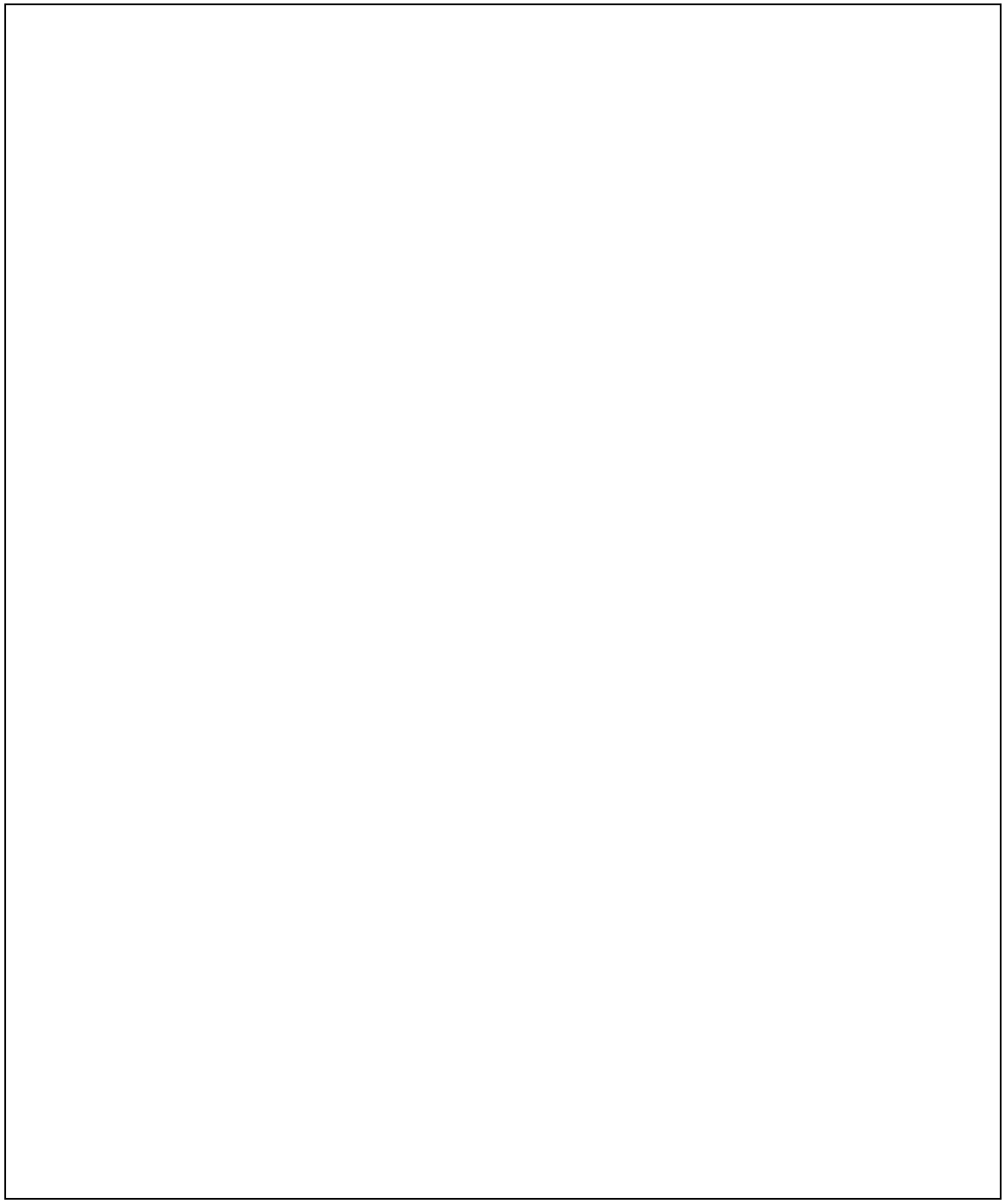
- *doLogin* uses *printf(username)*, i.e., the format string that will be used by *printf* corresponds to the username entered by the user. Therefore, by selecting a suitable format string for the entered username, you should be able to access the memory locations where the information you should find out is stored.
- The format tag *%p* can be used in the format string of *printf* to write the content of a pointer variable (i.e., the address stored in the variable) to *stdout*. This means that when using *%p* as the format string, *printf* interprets the corresponding parameter on the stack as an 8-byte (as we are working with a 64-bit OS) memory address and writes something like *0x7ffd426f01880* to *stdout* (leading 0-bytes are omitted). Likewise, using *%p*, e.g., four times as with *%p%p%p%p*, *printf* will interpret the next four parameters on the stack (32 bytes in total) as four memory addresses and writes them to *stdout*.
- Recall from the discussion at the beginning of this task that *printf* will always access the memory locations corresponding to the parameters on the stack, no matter whether the parameters were actually passed to *printf* or not. Also, recall that when using *printf* with a format string that contains more than five format tags, memory locations in the calling function are accessed. Therefore, when using *printf* with a format string such as *%p%p%p%p%p%p*, then the final *%p* will access the 8 bytes that are stored at the top of the stack frame of function *doLogin* (which calls *printf*). And by using even more *%p* format tags, guess what...
- As a final hint, the illustration below shows the situation on the stack when *printf* has been called (with stack frames for functions *printf*, *doLogin* and *main*). Note that this does not completely correspond to the «real» situation on the stack, as there are usually additional padding bytes used. But the relevant data that is stored on the stack is included in the illustration. It also shows the location where the format string parameter is stored (orange) and where the data that you should access during exploitation are stored (green).

	Content
stack frame of printf	...
	parameter 1: address of format string
	local variables (x bytes)
	stack canary
	stored rbp of doLogin function
	return address (stored rip)
stack frame of doLogin	parameter 2: passwords (8 bytes)
	parameter 1: usernames (8 bytes)
	local variable i (4 bytes)
	local variable username (8 bytes)
	local variable password (8 bytes)
	stored rbp of main function
	return address (stored rip)
stack frame of main	parameter 2: argv (8 bytes)
	parameter 1: argc (4 bytes)
	local variable command (8 bytes)
	local variable usernames (24 bytes)
	local variable passwords (24 bytes)
	stack canary
	stored rbp of caling function
	return address (stored rip)

Note that variables *usernames* and *passwords* (in stack frame of *main*) and *username* and *password* (in stack frame of *doLogin*) are stored in inverse declaration order. This reordering is done by *gcc* because stack protection is activated. The exact strategy used here is unclear, as just reordering two arrays (as with *usernames* and *passwords*) or two pointer variables (as with *username* and *password*) does not have an obvious security benefit (unlike the clear reordering benefit observed in task 2). In any case, this reordering is irrelevant for the vulnerability discussed in this task.

With the hints above, it should be possible for you to find and understand the vulnerability and, by exploiting it, to get the information stated above. Once you have managed to print the relevant parts of the stack to *stdout*, you may be able to identify the required information (*usernames* and *passwords*, `rbp` of the stack frame of *main*, stack canary) by comparing the output with the stack organization illustration above. However, to unambiguously identify the required information, it's recommended to also analyze the stack contents using *gdb*.

Document your findings in the following box. In particular, describe the vulnerability and how the login program can be used (by giving a specific example) to exploit the vulnerability and to get the required information. Also, clearly mark the required information in the output you get when exploiting the program. Providing a good answer will get you the fourth lab point.



Let’s summarize this task: The only mistake done by the programmer is that *printf(username)* is used instead of *printf("%s", username)*. This can easily happen in practice as the first version is an intuitive way to write a string to *stdout* (although *gcc* generates a warning, as you may have observed before when compiling the program). And using *printf* in this way works without any problems as long as there are no format tags in the string. However, if an attacker can control the string, as is the case here, he can read data from memory that shouldn’t be accessible.

Maybe you are wondering why you also had to get the base pointer and the stack canary because at first glance, these are not really valuable. But now assume that a similar format string vulnerability is part of a bigger server program that can be accessed remotely. And assume that this program contains another vulnerability that allows you to write beyond the end of a buffer on the stack. In this case, an attacker could first exploit the format string vulnerability to get the base pointer and the stack canary. Note that remote exploitation of format string vulnerabilities is possible if the server program accepts

user input and uses this data in a similar insecure way as in the example above with *printf*. In the case of remote exploitation, however, the problem is typically not an insecure usage of *printf*, but of a similar function such as *sprintf*, which writes the resulting string to a variable (instead of *stdout*), before sending it back to the client. Once the attacker knows the base pointer and the stack canary, he can exploit the other buffer overflow vulnerability to write beyond the stack canary, because knowing the stack canary value means he can make sure to overwrite it with the original value – so this protection mechanism is defeated and the attacker can now, e.g., successfully overwrite the stored return address. Also, assume the attacker wants to exploit the vulnerability to do a code injection attack, which means inserting code on the stack and overwriting the return address so that program execution jumps to the injected code when the program returns from the function. Assuming that ASLR is enabled, this is difficult as one cannot predict the address at which the code is injected. But as the attacker managed to learn the base pointer of the main stack frame by exploiting the format string vulnerability, he can now use this to calculate the effective memory locations that will be used during program executions on the stack (because also with ASLR enabled, the actual stack layout is always the same, it's just the address offset of the entire stack that changes whenever the program is started). Based on this, the attacker can make sure that program execution continues exactly at the location of the injected code when returning from the function. Note that this «story» is quite realistic. Modern buffer overflow exploitations often make use of multiple steps to first learn something about the internal memory organization or about some memory content, which can be used to defeat protection mechanisms, which then allows to do the actual attack.

Lab Points

In this lab, you can get **4 Lab Points**. To get them, you must demonstrate that you successfully solved tasks 2, 3 and 4:

- You get the first point for solving part 1 of task 2. Your answer (in general, write your answers directly into the boxes or into a separate document) must contain all elements as requested in Section 4.1. Also, you must demonstrate exploitation of the vulnerability and show that you can indeed read the content of the file *secret.txt*.
- If you have solved part 1 of task 2 correctly, you get the second point for a good answer to part 2 of task 2. It must contain all elements as requested in Section 4.2.
- You get the third and fourth points for solving tasks 3 and 4. Again, your answers must contain all elements as requested in Sections 5 and 6 and you must demonstrate that you can indeed exploit the vulnerabilities.

Handing in your solution can be done in two different ways:

- Show and explain your answers and demonstrate that you can exploit the vulnerabilities to the instructor in classroom, during the official lab lessons.
- Create screenshots or make a video of a terminal that shows that you can successfully exploit the vulnerabilities in tasks 2, 3 and 4, and send this together with your answers by e-mail to the instructor. Use *SecLab - Buffer Overflow - group X - name1 name2* as the e-mail subject, corresponding to your group number and the names of the group members. If the video is too big for an e-mail, place it somewhere (e.g., SWITCHdrive) and include a link to it in the e-mail.

7 Appendix

7.1 *gdb* commands

Below you find the most important commands of the GNU debugger. To use a program with the GNU debugger, the program must be compiled with the option -g, which adds necessary debugging information to the executable.

The debugger is started using the command *gdb <ExecutableName>*, where *ExecutableName* is the name of the program to debug.

`list` (or `l`)

Shows the next 10 source code lines. *list <LineNumber>* shows a few lines in front of and after the specified line. `list <FunctionName>, <LineNumber>` shows the lines of the specified function up to the specified line number.

`break` (or `b`)

Sets a breakpoint. *break <LineNumber>* sets a breakpoint at the specified line. *break <FunctionName>* sets a breakpoint at the beginning of the specified function.

`run <args>`

Runs the program. *<args>* are command line parameters that are passed to the program.

`delete <LineNumber>`

Deletes the breakpoint at the specified line. *delete* without arguments deletes all breakpoints.

`print <Variable>` (or `p`)

Shows the content of a variable. Using the address operator *(&<Variable>)* shows the address of a variable.

`continue` (or `c`)

Continues the program after is has stopped at a breakpoint.

`next` (or `n`)

Executes the next line. If it's a function call, the entire function is executed.

`step` (or `s`)

Just like *next*, but if the next line is a function call, the function is entered.

`backtrace` (or `bt`)

Shows the available stack frames.

`info`

Prints information about the running program. E.g., *info frame 0* shows the stack frame with number 0.

`x/<n>x <Address>` or `<Variable>`

Prints n double words starting from the specified address or the address of the specified variable.

`help <Command>`

Shows information about the specified command.

`quit`